METHOD AND APPARATUS FOR SPLITTING PACKETS IN A MULTITHREADED VLIW PROCESSOR

Cross-Reference to Related Applications

The present invention is related to United States Patent Application entitled "Method and Apparatus for Allocating Functional Units in a Multithreaded Very Large Instruction Word (VLIW) Processor," Attorney Docket Number (Berenbaum 7-2-3-3); United States Patent Application entitled "Method and Apparatus for Releasing Functional Units in a Multithreaded Very Large Instruction Word (VLIW) Processor," Attorney Docket Number (Berenbaum 8-3-4-4); and United States Patent Application entitled "Method and Apparatus for Identifying Splittable Packets in a Multithreaded Very Large Instruction Word (VLIW) Processor," Attorney Docket Number (Berenbaum 10-5-6-6), each filed contemporaneously herewith, assigned to the assignee of the present invention and incorporated by reference herein.

Field of the Invention

The present invention relates generally to multithreaded processors, and, more particularly, to a method and apparatus for splitting packets in such multithreaded processors.

Background of the Invention

Computer architecture designs attempt to complete workloads more quickly. A number of architecture designs have been proposed or suggested for exploiting program parallelism. Generally, an architecture that can issue more than one operation at a time is capable of executing a program faster than an architecture that can only issue one operation at a time. Most recent advances in computer architecture have been directed towards methods of issuing more than one operation at a time and thereby speed up the operation of programs. FIG. 1 illustrates a conventional microprocessor architecture 100. Specifically, the microprocessor 100 includes a program counter (PC) 110, a register set 120 and a number of functional units (FUs) 130-N. The redundant functional units (FUs) 130-1 through 130-N provide the illustrative microprocessor architecture 100 with sufficient hardware resources to perform a corresponding number of operations in parallel.

Cancel

An architecture that exploits parallelism in a program issues operands to more than one functional unit at a time to speed up the program execution. A number of architectures have been proposed or suggested with a parallel architecture, including superscalar processors, very long instruction word (VLIW) processors and multithreaded processors, each discussed below in conjunction with FIGS. 2, 4 and 5, respectively. Generally, a superscalar processor utilizes hardware at run-time to dynamically determine if a number of operations from a single instruction stream are independent, and if so, the processor executes the instructions using parallel arithmetic and logic units (ALUs). Two instructions are said to be independent if none of the source operands are dependent on the destination operands of any instruction that precedes them. A very long instruction word (VLIW) processor evaluates the instructions during compilation and groups the operations appropriately, for parallel execution, based on dependency information. A multithreaded processor, on the other hand, executes more than one instruction stream in parallel, rather than attempting to exploit parallelism within a single instruction stream.

A superscalar processor architecture 200, shown in FIG. 2, has a number of functional units that operate independently, in the event each is provided with valid data. For example, as shown in FIG. 2, the superscalar processor 200 has three functional units embodied as arithmetic and logic units (ALUs) 230-N, each of which can compute a result at the same time. The superscalar processor 200 includes a front-end section 208 having an instruction fetch block 210, an instruction decode block 215, and an instruction sequencing unit 220 (issue block). The instruction fetch block 210 obtains instructions from an input queue 205 of a single threaded instruction stream. The instruction sequencing unit 220 identifies independent instructions that can be executed simultaneously in the available arithmetic and logic units (ALUs) 230-N, in a known manner. The refine block 250 allows the instructions to complete, and also provides buffering and reordering for writing results back to the register set 240.

In the program fragment 310 shown in FIG. 3, instructions in locations L1, L2 and L3 are independent, in that none of the source operands in instructions L2 and L3 are dependent on the destination operands of any instruction that precedes them. When the program counter (PC) is set to location L1, the instruction sequencing unit 220 will look ahead in the instruction stream and detect that the instructions at L2 and L3 are independent, and thus all three can be issued simultaneously to the three available functional units 230-N. For a more detailed

discussion of superscalar processors, see, for example, James. E. Smith and Gurindar. S. Sohi, "The Microarchitecture of Superscalar Processors," Proc. of the IEEE (Dec. 1995), incorporated by reference herein.

As previously indicated, a very long instruction word (VLIW) processor 400, shown in FIG.4, relies on software to detect data parallelism at compile time from a single instruction stream, rather than using hardware to dynamically detect parallelism at run time. A VLIW compiler, when presented with the source code that was used to generate the code fragment 310 in FIG. 3, would detect the instruction independence and construct a single, very long instruction comprised of all three operations. At run time, the issue logic of the processor 400 would issue this wide instruction in one cycle, directing data to all available functional units 430-N. As shown in FIG. 4, the very long instruction word (VLIW) processor 400 includes an integrated fetch/decode block 420 that obtains the previously grouped instructions 410 from memory. For a more detailed discussion of very long instruction word (VLIW) processors, see, for example, Burton J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," SPIE Real Time Signal Procesing IV, 241-248 (1981), incorporated by reference herein.

One variety of VLIW processors, for example, represented by the Multiflow architecture, discussed in Robert P. Colwell et al., "A VLIW Architecture for a Trace Scheduling Compiler," IEEE Transactions on Computers (August 1988), uses a fixed-width instruction, in which predefined fields direct data to all functional units 430-N at once. When all operations specified in the wide instruction are completed, the processor issues a new, multi-operation instruction. Some more recent VLIW processors, such as the C6x processor commercially available from Texas Instruments, of Dallas, TX and the EPIC IA-64 processor commercially available from Intel Corp, of Santa Clara, CA, instead use a variable-length instruction packet, which contains one or more operations bundled together.

A multithreaded processor 500, shown in FIG. 5, gains performance improvements by executing more than one instruction stream in parallel, rather than attempting to exploit parallelism within a single instruction stream. The multithreaded processor 500 shown in FIG. 5 includes a program counter 510-N, a register set 520-N and a functional unit 530-N, each dedicated to a corresponding instruction stream N. Alternate implementations of the

multithreaded processor 500 have utilized a single functional unit 530, with several register sets 520-N and program counters 510-N. Such alternate multithreaded processors 500 are designed in such a way that the processor 500 can switch instruction issue from one program counter/register set 510-N/520-N to another program counter/register set 510-N/520-N in one or two cycles. A long latency instruction, such as a LOAD instruction, can thus be overlapped with shorter operations from other instruction streams. The TERA MTA architecture, commercially available from Tera Computer Company, of Seattle, WA, is an example of this type.

An extension of the multithreaded architecture 500, referred to as Simultaneous Multithreading, combines the superscalar architecture, discussed above in conjunction with FIG. 2, with the multithreaded designs, discussed above in conjunction with FIG. 5. For a detailed discussion of Simultaneous Multithreading techniques, see, for example, Dean Tullsen et al., “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” Proc. of the 22nd Annual Int’l Symposium on Computer Architecture, 392-403 (Santa Margherita Ligure, Italy, June 1995), incorporated by reference herein. Generally, in a Simultaneous Multithreading architecture, there is a pool of functional units, any number of which may be dynamically assigned to an instruction which can issue from any one of a number of program counter/register set structures. By sharing the functional units among a number of program threads, the Simultaneous Multithreading architecture can make more efficient use of hardware than that shown in FIG. 5.

While the combined approach of the Simultaneous Multithreading architecture provides improved efficiency over the individual approaches of the superscalar architecture or the multithreaded architecture, Simultaneous Multithreaded architectures still require elaborate issue logic to dynamically examine instruction streams in order to detect potential parallelism. In addition, when an operation takes multiple cycles, the instruction issue logic may stall, because there is no other source of operations available. Conventional multithreaded processorss issue instructions from a set of instructions simultaneously, with the functional units designed to accommodate the widest potential issue. A need therefore exists for a multithreaded processor architecture that does not require a dynamic determination of whether or not two instruction streams are independent. A further need exists for a multithreaded architecture that provides

simultaneous multithreading. Yet another need exists for a method and apparatus that improve the utilization of processor resources for each cycle.

Summary of the Invention

Generally, a method and apparatus are disclosed for processing instructions in a multithreaded very large instruction word (VLIW) processor. The present invention combines the techniques of conventional very long instruction word (VLIW) architectures and conventional multithreaded architectures. The combined architecture of the present invention reduces execution time within an individual program, as well as across a workload. The present invention utilizes instruction packet splitting to recover some efficiency lost with conventional multithreaded architectures. Instruction packet splitting allows an instruction bundle to be partially issued in one cycle, with the remainder of the bundle issued during a subsequent cycle. Thus, the present invention provides greater utilization of hardware resources (such as the functional units) and a lower elapsed time across a workload comprising multiple threads.

Instruction packet splitting increases throughput across all instruction threads, and reduces the number of cycles that the functional units rest idle. The allocation hardware of the present invention assigns as many instructions from each packet as will fit on the available functional units, rather than allocating all instructions in an instruction packet at one time. Those instructions that cannot be allocated to a functional unit are retained in a ready-to-run register. On subsequent cycles, instruction packets in which all instructions have been issued to functional units are updated from their thread's instruction stream, while instruction packets with instructions that have been held are retained. The functional unit allocation logic can then assign instructions from the newly-loaded instruction packets as well as instructions that were not issued from the retained instruction packets.

The present invention utilizes a compiler to detect parallelism in a multithreaded processor architecture. Thus, a multithreaded VLIW architecture is disclosed that exploits program parallelism by issuing multiple instructions, in a similar manner to single threaded VLIW processors, from a single program sequencer, and also supporting multiple program sequencers, as in simultaneous multithreading but with reduced complexity in the issue logic, since a dynamic determination is not required.

A more complete understanding of the present invention, as well as further features and advantages of the present invention, will be obtained by reference to the following detailed description and drawings.

Brief Description of the Drawings

FIG. 1 illustrates a conventional generalized microprocessor architecture;

FIG. 2 is a schematic block diagram of a conventional superscalar processor architecture;

FIG. 3 is a program fragment illustrating the independence of operations;

FIG. 4 is a schematic block diagram of a conventional very long instruction word (VLIW) processor architecture;

FIG. 5 is a schematic block diagram of a conventional multithreaded processor;

FIG. 6 illustrates a multithreaded VLIW processor in accordance with the present invention;

FIG. 7A illustrates a conventional pipeline for a multithreaded processor;

FIG. 7B illustrates a pipeline for a multithreaded processor in accordance with the present invention;

FIG. 8 is a schematic block diagram of an implementation of the allocate stage of FIG. 7B;

FIG. 9 illustrates the execution of three threads TA-TC for a conventional multithreaded implementation, where threads B and C have higher priority than thread A; and

FIGS. 10A and 10B illustrate the operation of instruction packet splitting in accordance with the present invention.

Detailed Description

FIG. 6 illustrates a Multithreaded VLIW processor 600 in accordance with the present invention. As shown in FIG. 6, there are three instruction threads, namely, thread A (TA), thread B (TB) and thread C (TC), each operating at instruction number *n*. In addition, the illustrative Multithreaded VLIW processor 600 includes nine functional units 620-1 through 620-9, which can be allocated independently to any thread TA-TC. Since the number of instructions

across the illustrative three threads TA-TC is nine and the illustrative number of available functional units 620 is also nine, then each of the instructions from all three threads TA-TC can issue their instruction packets in one cycle and move onto instruction *n+1* on the subsequent cycle.

It is noted that there is generally a one-to-one correspondence between instructions and the operation specified thereby. Thus, such terms are used interchangeably herein. It is further noted that in the situation where an instruction specifies multiple operations, it is assumed that the multithreaded VLIW processor 600 includes one or more multiple-operation functional units 620 to execute the instruction specifying multiple operations. An example of an architecture where instructions specifying multiple operations may be processed is a complex instruction set computer (CISC).

The present invention allocates instructions to functional units to issue multiple VLIW instructions to multiple functional units in the same cycle. The allocation mechanism of the present invention occupies a pipeline stage just before arguments are dispatched to functional units. Thus, FIG. 7A illustrates a conventional pipeline 700 comprised of a fetch stage 710, where a packet is obtained from memory, a decode stage 720, where the required functional units and registers are identified for the fetched instructions, and an execute stage 730, where the specified operations are performed and the results are processed.

Thus, in a conventional VLIW architecture, a packet containing up to K instructions is fetched each cycle (in the fetch stage 710). Up to K instructions are decoded in the decode stage 720 and sent to (up to) K functional units (FUs). The registers corresponding to the instructions are read, the functional units operate on them and the results are written back to registers in the execute stage 730. It is assumed that up to three registers can be read and up to one register can be written per functional unit.

FIG. 7B illustrates a pipeline 750 in accordance with the present invention, where an allocate stage 780, discussed further below in conjunction with FIG. 8, is added for the implementation of multithreaded VLIW processors. Generally, the allocate stage 780 determines how to group the operations together to maximize efficiency. Thus, the pipeline 750 includes a fetch stage 760, where up to N packets are obtained from memory, a decode stage 770, where the functional units and registers are identified for the fetched instructions (up to N*K instructions),

an allocate stage 780, where the appropriate instructions are selected and assigned to the FUs, and an execute stage 790, where the specified operations are performed and the results are processed.

In the multithreaded VLIW processor 600 of the present invention, up to N threads are supported in hardware. N thread contexts exist and contain all possible registers of a single thread and all status information required. A multithreaded VLIW processor 600 has M functional units, where M is greater than or equal to K. The modified pipeline stage 750, shown in FIG. 7B, works in the following manner. In each cycle, up to N packets (each containing up to K instructions) are fetched at the fetch stage 760. The decode stage 770 decodes up to N*K instructions and determines their requirements and the registers read and written. The allocate stage 780 selects M out of (up to) N*K instructions and forwards them to the M functional units. It is assumed that each functional unit can read up to 3 registers and write one register. In the execute stage 790, up to M functional units read up to 3*M registers and write up to M registers.

The allocate stage 780 selects for execution the appropriate M instructions from the (up to) N * K instructions that were fetched and decoded at stages 760 and 770. The criteria for selection are thread priority or resource availability or both. Under the thread priority criteria, different threads can have different priorities. The allocate stage 780 selects and forwards the packets (or instructions from packets) for execution belonging to the thread with the highest priority according to the priority policy implemented. A multitude of priority policies can be implemented. For example, a priority policy for a multithreaded VLIW processor supporting N contexts (N hardware threads) can have N priority levels. The highest priority thread in the processor is allocated before any other thread. Among threads with equal priority, the thread that waited the longest for allocation is preferred.

Under the resource availability criteria, a packet (having up to K instructions) can be allocated only if the resources (functional units) required by the packet are available for the next cycle. Functional units report their availability to the allocate stage 780.

FIG. 8 illustrates a schematic block diagram of an implementation of the allocate stage 780. As shown in FIG. 8, the hardware needed to implement the allocate stage 780 includes a priority encoder 810 and two crossbar switches 820, 830. Generally, the priority encoder 810 examines the state of the multiple operations in each thread, as well as the state of

the available functional units. The priority encoder 810 selects the packets that are going to execute and sets up the first crossbar switch 820 so that the appropriate register contents are transferred to the functional units at the beginning of the next cycle. The output of the priority encoder 810 configures the first crossbar switch 820 to route data from selected threads to the appropriate functional units. This can be accomplished, for example, by sending the register identifiers (that include a thread identifier) to the functional units and letting the functional units read the register contents via a separate data network and using the crossbar switch 810 to move the appropriate register contents to latches that are read by the functional units at the beginning of the next cycle

Out of the N packets that are fetched by the fetch stage 760 (FIG. 7B), the priority encoder 810 selects up to N packets for execution according to priority and resource availability. In other words, the priority encoder selects the highest priority threads that do not request unavailable resources for execution. It then sets up the first crossbar switch 810. The input crossbar switch 810 routes up to 3K*N inputs to up to 3*M outputs. The first crossbar switch 810 has the ability to transfer the register identifiers (or the contents of the appropriate registers) of each packet to the appropriate functional units.

Since there are up to N threads that can be selected in the same cycle and each thread can issue a packet of up to K instructions and each instruction can read up to 3 registers there are 3K*N register identifiers to select from. Since there are only M functional units and each functional unit can accept a single instruction, there are only 3M register identifiers to be selected. Therefore, the crossbar switch implements a 3K*N to 3M routing of register identifiers (or register contents).

The output crossbar switch 830 routes M inputs to N*M or N*K outputs. The second crossbar switch 830 is set up at the appropriate time to transfer the results of the functional units back to the appropriate registers. The second crossbar switch 830 can be implemented as a separate network by sending the register identifiers (that contain a thread identifier) to the functional units. When a functional unit computes a result, the functional unit routes the result to the given register identifier. There are M results that have to be routed to up to N threads. Each thread can accept up to K results. The second crossbar switch 830 routes M results to N*K possible destinations. The second crossbar switch 830 can be implemented as M

buses that are connected to all N register files. In this case, the routing becomes M results to N*M possible destinations (if the register files have the ability to accept M results).

In a conventional single-threaded VLIW architecture, all operations in an instruction packet are issued simultaneously. There are always enough functional units available to issue a packet. When an operation takes multiple cycles, the instruction issue logic may stall, because there is no other source of operations available. In a multithreaded VLIW processor in accordance with the present invention, on the other hand, these restrictions do not apply.

FIG. 9 illustrates the execution of three threads TA-TC for a conventional multithreaded implementation (without the benefit of the present invention), where threads B and C have higher priority than thread A. Since thread A runs at the lowest priority, its operations will be the last to be assigned. As shown in FIG. 9, five functional units 920 are assigned to implement the five operations in the current cycle of the higher priority threads TB and TC. Thread A has four operations, but there are only two functional units 920 available. Thus, thread A stalls for a conventional multithreaded implementation.

In order to maximize throughput across all threads, and minimize the number of cycles that the functional units rest idle, the present invention utilizes instruction packet splitting. Instead of allocating all operations in an instruction packet at one time, the allocation hardware 780, discussed above in conjunction with FIG. 8, assigns as many operations from each packet as will fit on the available functional units. Those operations that will not fit are retained in a ready-to-run register 850 (FIG. 8). On subsequent cycles, instruction packets in which all operations have been issued to functional units are updated from their thread's instruction stream, while instruction packets with operations that have been held are retained. The functional unit allocation logic 780 can then assign operations from the newly-loaded instruction packets as well as operations that were not issued from the retained instruction packets.

The operation of instruction packet splitting in accordance with the present invention is illustrated in FIGS. 10A and 10B. In FIG. 10A, there are three threads, each with an instruction packet from location *n* ready to run at the start of cycle *x*. Thread A runs at the lowest priority, so its operations will be the last to be assigned. Threads B and C require five of the seven available functional units 1020 to execute. Only two functional units 1020-2 and 1020-6

remain, so only the first two operations from thread A are assigned to execute. All seven functional units 1020 are now fully allocated.

At the completion of cycle x, the instruction packets for threads B and C are retired. The instruction issue logic associated with the threads replaces the instruction packets with those for address $n+1$, as shown in FIG. 10B. Since the instruction packet for thread A is not completed, the packet from address n is retained, with the first two operations marked completed. On the next cycle, $x+1$, illustrated in FIG. 10B, the final two operations from thread A are allocated to functional units, as well as all the operations from threads B and C.

Thus, the present invention provides greater utilization of hardware resources (i.e., the functional units 1020) and a lower elapsed time across a workload comprising the multiple threads.

It is to be understood that the embodiments and variations shown and described herein are merely illustrative of the principles of this invention and that various modifications may be implemented by those skilled in the art without departing from the scope and spirit of the invention.